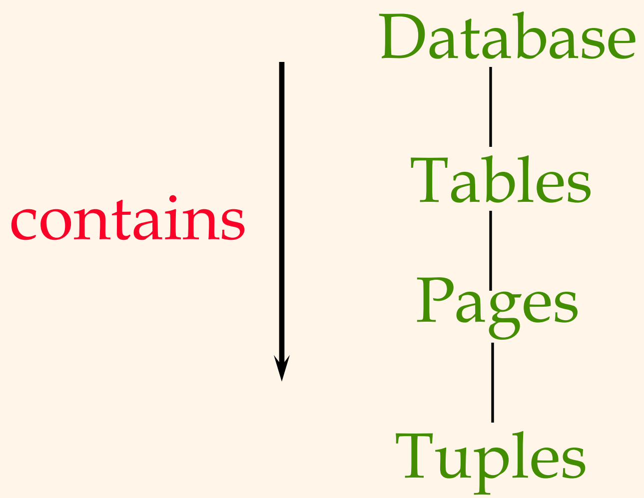
Locking wrap-up
Nonlocking Concurrency Control

Reading

- [RG] 17.1-17.5, 17.6

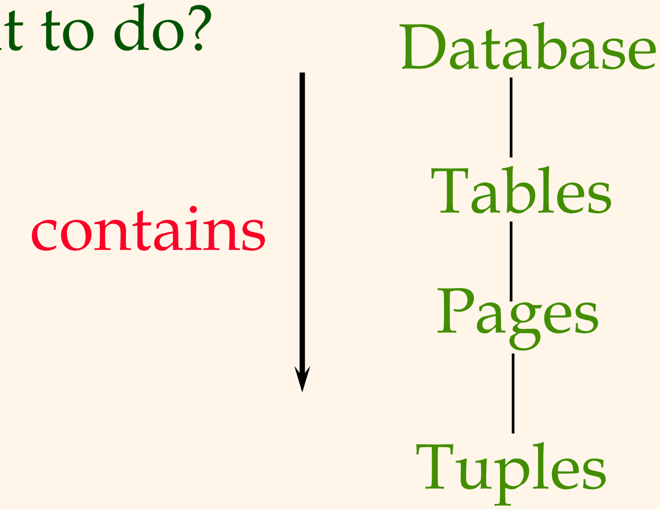
Multiple-Granularity Locks

- Suppose we have a hierarchy on data "containers"
- Can lock at different levels



Multiple-Granularity Locks

- If we need to modify lots of data in the table, faster to lock whole table rather than each tuple individually
  - OTOH this blocks everyone else from accessing the file
  - What to do?



Solution: New Lock Modes, Protocol

- Allow locks at all levels of hierarchy
- Need to ensure locks granted in a consistent manner
  - eg. don't give someone a write lock on a tuple when someone else has a read lock on the whole table

Solution: New Lock Modes, Protocol

- A special protocol using new "intention" locks.
- Before locking an item, transaction must set "intention locks" on all its ancestors.
  - IS if wants to read and get S lock
  - IX if wants to write and get X lock

Multiple Granularity Lock Protocol

- Transactions start from the root of the hierarchy.
- To get S lock, must hold IS on parent node.
- To get X lock, must hold on IX parent node.
- Holding a lock on a node means I implicitly hold the same lock on its descendants in hierarchy

What locks can be granted simultaneously on an object?

	IS	IX	S	X
IS	√	√	√	
IX	√	√		
S	√		√	
X				

SIX locks

- Often want to read whole table and modify a few tuples
  - So, on the table, want S lock plus IX lock
  - SIX lock is a handy shortcut
- These locks can be used in conjunction with 2PL to ensure serializability.

Locking summary

- 2PL and variants
- Deadlock detection and prevention
- Phantom problem and solutions
- Custom locking for a data structure (B+-trees)
  - Violates 2PL but still correct due to unique access patterns in tree
- Multiple granularity locking

Optimistic CC

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - These overheads occur even if conflicts are rare
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before commit.

Optimistic CC

- Transactions have three phases:
  - READ: read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

Validation

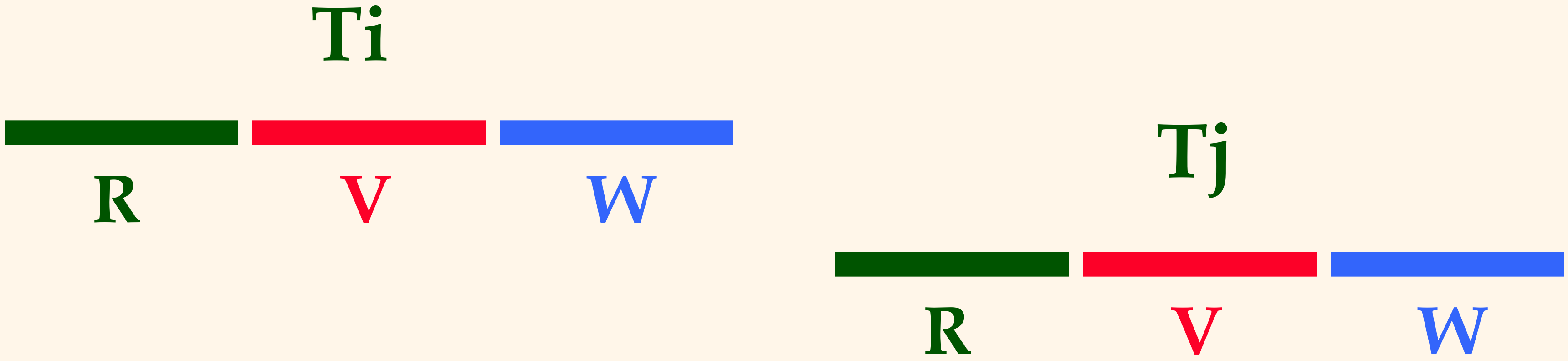
- Test conditions that are sufficient to ensure that no conflict occurred.
- Each transaction is assigned a numeric id (eg timestamp)
- Ids assigned at beginning of validation phase
  - Idea: will serialize transactions in that order
- ReadSet(Ti): Set of objects read by Ti.
- WriteSet(Ti): Set of objects modified by Ti.

Validation

- Want to detect if a pair of transactions Ti and Tj may have conflicted
  - If so, one needs to be aborted and restarted
- Ti and Tj are ok (non-conflicting), as long as certain conditions hold

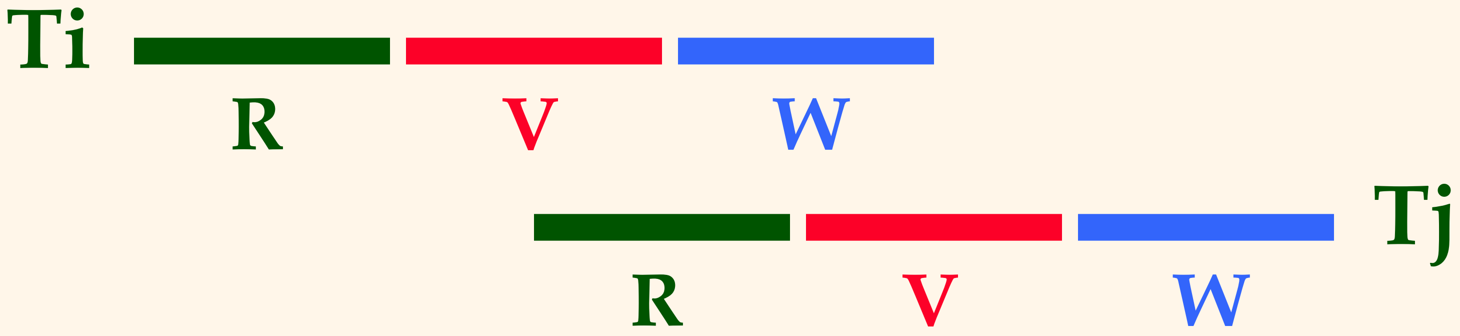
OK scenario 1

- Ti completes before Tj begins.



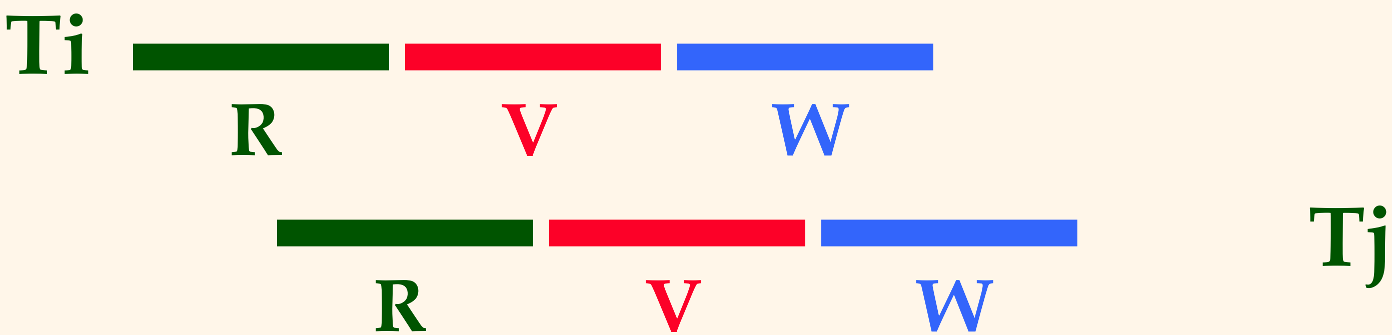
OK scenario 2

- Ti completes before Tj begins its Write phase
- WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.



OK scenario 3

- Ti completes Read phase before Tj does
- WriteSet(Ti) $\cap$ ReadSet(Tj) is empty
- WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.



OCC protocols

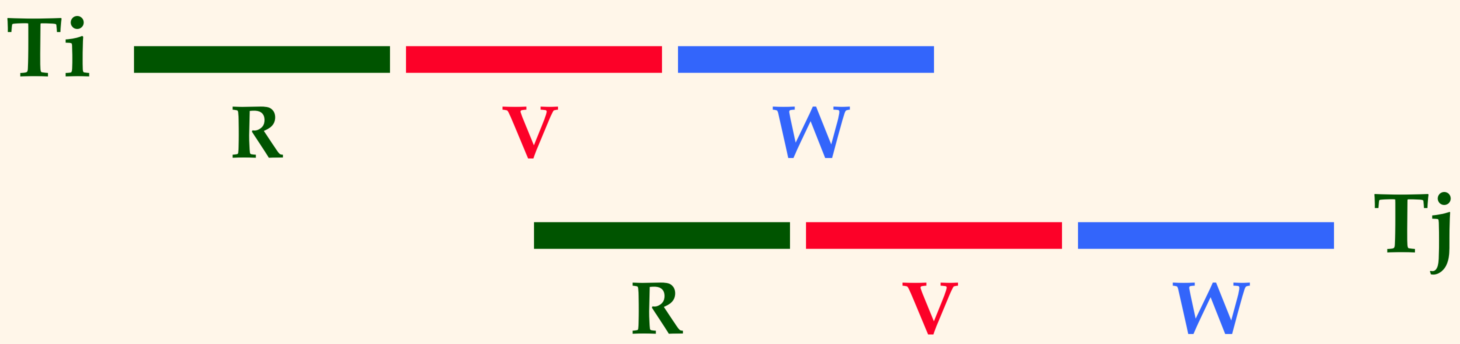
- Ensure that all possible events/executions in the system fall into one of the three "safe" cases
- Can be overly strong yet still correct
  - E.g. could allow only case 1 – serial execution

Validation/Write

- Allow scenarios 1 and 2
- Before commit, every transaction T goes through a combined Validation/Write phase
- Only one transaction can be in this phase at a time (so don't need to worry about scenario 3; write phases never overlap)

OK scenario 2

- Ti completes before Tj begins its Write phase
- WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.



Validation/Write

- For a given transaction T, find all transactions T' that committed after T started (and could overlap with it)
- Check that read set of T does not intersect the write set of any such T'
  - If so, apply writes and commit, else abort

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per transaction
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
- Optimistic CC restarts transactions that fail validation.
  - Work done so far is wasted
  - In a high-contention workload, OCC may not be the best choice

Multiversion Concurrency Control

R1(A) W1(A) R2(A) W2(B) R1(B) W1(C)

- Not conflict-serializable
- Intuition behind problem: R1(B) is "just" too late
  - If we had kept the old version of B around, could just give it to Transaction 1 instead of current version
- This is the main idea behind MVCC

MVCC

- System keeps several versions of each data item
- When a transaction writes a data item, it creates a new version rather than overwriting
- When a transaction reads a data item, the version visible to the read is determined by the protocol used (several options)
- Maintaining versions can be nontrivial and comes with its own extra cost, of course

Timestamp MVCC protocol

- Each transaction gets a timestamp when it arrives in the system
- Idea: serialize the transactions in the order of timestamps
- If transaction i wants to read object A, system shows it the version of A written by the largest k such that k < i
  - assume transactions don't read objects after having written them, protocol a bit more involved if not so

Timestamp MVCC protocol

- When transaction i wants to write (a new version of) object A, perform a check
- Has some transaction j, j > i, already read version k of A for some k < i?
  - Should have read i's version instead!
  - If so, i is aborted and restarted with a new (higher) timestamp

MVCC *and aborts*

- The protocol we just saw guarantees serializability
- If you additionally want nice abort-related properties like recoverability, ACA or strictness, need to augment the protocol to enforce them.
- Eg. for recoverability, delay commit of any transaction T until all transactions T has "read from" have committed

Snapshot Isolation

- A different way to use versions
- Version visible to read by transaction T1 is **last committed version as of T1's start time**
  - "snapshot" of DB as of T1's start time

Snapshot Isolation

- First Committer Wins rule needed for correctness
- If two transactions whose executions overlap in time write to the same data item, one must abort
- In practice, can abort T1 as soon as we detect that some T2 has committed, where T2 and T1 wrote to the same item

Snapshot Isolation

- Easy to implement
- Reads never block
- But SI permits schedules which are not actually serializable
- *Write skew* anomalies

Write skew example

```
create table a ( x int );
create table b ( x int );
```

t	Transaction T1	Transaction T2
1	`Insert into a select count(*) from b;`	
2		`Insert into b select count(*) from a;`
3	`Commit;`	
4		`Commit;`